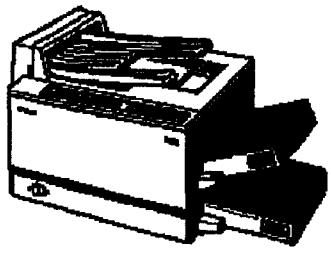

The PostScript™ Guest Column

Mitch Bradley

Forth in PostScript

[Editor's Note: this article shows the flexibility of Forth and PostScript. It also should help users of either language to understand the other better. *Copyright (c) 1990 Bradley Forthware. Permission is hereby granted to use this code for any purpose, provided that this notice remains attached.* *Address correspondence to: Bradley Forthware, P.O. Box 4444, Mountain View, CA 94040.]*

These comments describe the implementation in PostScript of a subset of Forth. They assume that the reader knows Forth, but doesn't know PostScript. This does not presume to be a complete implementation of Forth, but instead an intellectual exercise, a "proof of concept", and a brief introduction for Forth programmers to some interesting aspects of the PostScript language. Despite its deficiencies, Forth programmers might still find this program useful as a way to control laser printers using a syntax that is more familiar to them. While I personally feel that PostScript syntax is superior to Forth syntax in many ways, my long experience with Forth allows me to "think" in Forth more readily than in PostScript.

This program supports Forth colon definition syntax, the **IF .. ELSE .. THEN** conditionals, and **CREATE .. DOES>** (however, "does>" had to be spelled without the ">" because PostScript doesn't allow the special characters **()[]{}<>/%** in names). Some Forth stack and arithmetic operators are provided; others are missing. Most of the missing ones could be added with little effort. PostScript procedures may be executed from within the Forth environment and compiled into definitions, so the Forth syntax may be used to write graphics programs using the PostScript graphics operators.

It is a tribute to the power and beauty of the PostScript language that I was able to write the following code (excluding the commentary) in about four hours, while still learning PostScript. To use, "run" this file and then execute **QUIT** (this is the Forth "quit", which really should have been named "restart", not the PostScript "quit").

Implementation Details

Define a new procedure "-def" which is like the normal defining procedure "def", except that the operands are reversed. "def" associates a keyword with a procedure or a value; "def" performs the functions of the Forth words : (colon), **CONSTANT** and **VARIABLE** .

```
/-def  { exch def } def   % proc key --
```

Create a new dictionary (like a Forth vocabulary) named "immeddict", with space for 20 keywords (names). This dictionary will be used to contain the "immediate" words for the Forth implementation.

```
/immeddict  20 dict   def  % -- dict
```

Create a new dictionary "forthdict", with space for 300 keywords (names). This dictionary will be used to contain the "non-immediate" words for the Forth implementation.

```
/forthdict  300 dict  def  % -- dict
```

Create a procedure named "forth" which will push the "forthdict" dictionary on the PostScript dictionary stack (search order stack). The procedure "begin" pops a dictionary object from the data stack and pushes it on the dictionary stack.

```
/forth  { forthdict begin }  def %  --
```

Push the forthdict dictionary on the PostScript dictionary stack. This causes that dictionary to be searched first, and also causes new definitions to be placed in that dictionary.

```
forth
```

Define some Forth names for common operations. These operations have direct PostScript equivalents, usually with different names. Here we alias them to the equivalent Forth names for the convenience of us Forth programmers. The "load" operator is for increased efficiency; it forces the names to be looked up at compile time and be replaced by the actual executable object. Otherwise, the alias words would be looked-up at run time, every time that the word is executed. This run-time lookup is the default PostScript behavior, in contrast to Forth which defaults to compile-time binding. The "load" and "bind" PostScript operators force compile-time binding.

Note that the "/" operator must be specified differently from the rest. "/" has special syntactic meaning in PostScript; it signifies that the rest of the name is to be interpreted as a literal keyword, instead of being executed.

```
/dup     /dup   load def  %  obj -- obj obj
/swap    /exch  load def  %  o1 o2 -- o2 o1
/drop    /pop   load def  %  obj --
/+       /add   load def  %  n1 n2 -- n1+n2
/-       /sub   load def  %  n1 n2 -- n1-n2
/*       /mul   load def  %  n1 n2 -- n1*n2
(/) cvn  /idiv  load def  %  n1 n2 --
                                  int(n1/n2)
```

Now that we have redefined "/" , you may wonder why it can still be used in its normal way in the following code. The answer is that "/" has meaning to the lexical scanner; it is not "looked up in the dictionary". By analogy, the character "." in some Forths has special

meaning to the number parser (signifying a double number), even though there is also a "." entry in the dictionary with a different meaning.

Stack Manipulation Operations

Here are some Forth names for common stack operations. Note the PostScript operator "roll". It is quite powerful, rotating "n" stack cells by "m" positions, in either direction. The Forth **ROLL** is a special case of the PostScript "roll". The Forth version of **ROLL** could be defined as:

```
/forth-roll  { -1 roll }  def

/over     { 1 index }     def  % o1 o2 --
                                  o1 o2 o1
/nip      { swap drop }   def  % o1 o2 -- o2
/2dup     { 2 copy }      def  % o1 o2 --
                                  o1 o1 o1 o2
/rot      { 3 -1 roll }   def  % o1 o2 o3 --
                                  o2 o3 o1
/-rot     { 3  1 roll }   def  % o1 o2 o3 --
                                  o3 o1 o2
```

Booleans and Conditionals

Here are a couple of handy Forth words. Their definition also illustrates two PostScript features: nested procedures and the "if" operator. The innermost set of braces defines a nested procedure. The "if" operator is completely postfix; its stack diagram is:

★ boolean proc --

The procedure object is executed if the flag is true, and dropped if the boolean value is false. PostScript also has "ifelse"; its stack diagram is:

★ boolean true-proc false-proc --

```
/max  { 2dup lt  { swap }  if  drop  }  def
                                % n1 n2 -- max
/min  { 2dup gt  { swap }  if  drop  }  def
                                % n1 n1 -- min
```

Here are the run-time words which the Forth implementation compiles for the **IF .. THEN** and **IF .. ELSE .. THEN** constructs. "0 ne" converts a number into a flag. Forth conditionals use particular values of integers to signify truth values. PostScript has a separate boolean data type, and a "typecheck" run-time error will result if a conditional operator encounters a number on the stack instead of a boolean.

```
/pif  { swap 0 ne swap if } bind def
                                % n proc --
/pifelse { rot 0 ne -rot ifelse } bind def
                        % n t-proc f-proc --
```

Handling the Input Stream

Use this handy constant:

```
/bl  32  def  %  -- n
```

The Forth text interpreter uses "STATE" to distinguish compilation from interpretation. In this case, it is more like a **VALUE** than a **VARIABLE**, because it does not need to be followed by @ (fetch) in PostScript.

```
/state  0  def  % -- n
```

Read an edited line from the keyboard, placing it into the string object on the stack, returning the filled substring of that string:

```
/expect  {  % string -- substring
  (%lineedit) (r) file /ifd -def  % string
  ifd swap readstring drop      % substring
  ifd closefile                 % substring

  dup length 1 sub 0 max % workaround for
                         % bug in Imagen
  0 swap getinterval     % "UltraScript"
                         % emulator
  } def
```

Read an edited line; put it in a string named "tib":

```
/query  { 80 string  expect  /tib -def  }
  def                   % --
```

Parse a delimited string from the input string "source", updating the source string to contain the remainder of the input after the string that was parsed:

```
/word  {                % delim -- string
  /ws 1 string  def     % delim
  ws 0 rot put
  source ws  search
  { nip swap } { () }  ifelse  /source -def
  } def
```

Convert a numeric string to an integer:

```
/number  { cvi }  def    % string -- int
```

Search for a string in all the dictionaries on the dictionary stack:

```
/find  {  % str -- str false  |  proc true
  dup cvn where
        % str dict true  |  str false
  { drop cvn cvx true  }
  { false }
  ifelse  % str false  |  proc true
 } def
```

Compilation and Interpretation

Here's how the Forth interpreter handles a word when it is in compile state. Note that a compiled word is not "comma'ed into the dictionary"; instead, a procedure object for that word is left on the stack for later inclusion in a new procedure array. "Immediate" words are those which are located in the "immeddict" dictionary:

```
/do-compile  {        % str -- [ proc | n ]
  immeddict over cvn known
                      % str flag
```

Execute immediate words:

```
{ immeddict swap cvn get cvx exec }
                      % <nothing>
```

Leave procedures on stack for later, leave number if word not found:

```
{ find not  { number } if  }
                      % [ proc | n ]
  ifelse              % proc | n | <nothing>
  } def
```

This is how the Forth interpreter handles a word in interpret state. If the word is found, it is executed. Otherwise, it is converted to a number. If the numeric conversion fails, PostScript will automatically signal an error, which we will intercept at a higher level:

```
/do-interpret  {
  find  { exec }  { number } ifelse
  } def
```

Interpret a line of text. Note that PostScript loops are postfix too. "loop" executes the preceding procedure until "exit" is executed. "exit" terminates execution of the nearest enclosing "loop":

```
/interpret  {         % -- ??
  tib /source -def    % Set up the text line
  {
  bl word             % string
  dup length  0 eq    { drop exit }  if
                      % string
  state 0 ne { do-compile } { do-interpret}
  ifelse
  }
  loop
  } def
```

Okay, here it is: the top level of the Forth text interpreter. "stopped" catches error signals (PostScript error signals are sort of like Forth **ABORT** except that the program can catch and handle them if it chooses). "stopped" prevents errors from kicking you out of the Forth interpreter and back into the PostScript interpreter. "(OK)" is the string **OK** , which is displayed as the user prompt:

```
/quit {
  {
  query
  { interpret } stopped          % error?
  { handleerror } { ( OK) print }  if
  }
  loop
  } def
```

Some Stack and Debugging Words

Here are some more handy Forth words:

```
/.    /= load  def                % obj --
/.s   /pstack load  def           % ?? -- ??
/depth  { count } def
```

And here are some debugging tools; "words" displays the words in the dictionary. "psee" and "see" are decompilers. "psee" takes a keyword from the stack and decompiles the object or procedure it refers to. "see" takes a name from the input stream and decompiles it. Note how simple these are to implement. That is because PostScript has a function to enumerate the words in a dictionary ("forall") and also a general purpose display word "==" which will basically decompile anything. Forth really ought to at least have a standard way to enumerate the words in a vocabulary!

```
/words { currentdict { pop = } forall } def
/psee  {
  dup where  { swap get == }  { drop (?)
  print }  ifelse
  } def
/see    { bl word cvn psee }  def
```

Some Forth-style literals:

```
/'   { bl word cvx }     def % -- proc
/"   { (") 0 get word } def % -- string
/"" { bl word cvn }     def % -- name
```

Data Space Operations

Create a big array called "memory" from which we "allot" space. Fetch and store access elements of this array. From here on out, "adr" really means an index into this data array, not a real machine address:

```
/limit    10000  def              % -- adr
/memory   limit array  def        % -- array
/here     0  def                  % -- adr
/allot { here + /here -def } def % size --
/@ { memory swap get }       def % adr -- obj
/! { memory swap rot put } def % obj adr --
/, { here  1 allot  !  }  def % obj --
```

Forth Defining Words

Here's where it really starts to get interesting! We start to build some defining words. The easiest defining word is "constant". It's trivial because PostScript can associate a number with a keyword, and that's just what a constant is. By the way, "cvn" converts a string to a number:

```
/constant  { bl word cvn swap def }  def
```

Now we deal with colon definitions. The PostScript equivalent of a Forth colon definition is an "executable array". The array is an important data type in PostScript. Arrays contain objects and basically everything in PostScript is an object. Numbers are objects, as are strings, procedures, arrays, keywords (names), and dictionaries. The objects contained in an array can be of any type, and different types may be mixed within the same array. A PostScript procedure is an array which is marked as executable. A non-executable array is simply pushed on the stack when it is encountered. An executable array is executed. An array can be converted from non-executable to executable at any time.

We compile a Forth colon definition by building an array containing the procedures the comprise the definition. When the colon definition is terminated with ";", we close the array and mark it executable. The basic tools we use for creating executable arrays are "+level" and "-level". "+level" begins the array, and "-level" finishes it. Executable arrays may be nested; we need this for implementing control structures within colon definitions. The nesting depth is indicated by the value of "state", which is incremented by "+level" and decremented by "-level". When state is non-zero, we are in compile state and each word is added to the array we are building, otherwise we are in interpret state, and each word is executed as it is encountered:

```
/+level  { /state  state 1 +  def [  } def
/-level  { ] cvx  bind  /state  state 1 -
  def  }  def
```

Here's ":". We remember the name on the stack and start building an array. When the matching ";" is encountered, we will finish the array, make it executable, and attach the name to it:

```
/:  { bl word cvn  +level  }  def
```

CREATE ... DOES> is pretty tricky. The basic idea is that **CREATE** makes a word of the form:

★ { <address> noop noop }

<address> is the value of **HERE** at the time the word was created. If the new word is not later modified by **DOES>** , then the noop's will be executed (not doing anything) and the address will be left on the stack. If there is a **DOES>** , then the first "noop" will be replaced with a procedure which implements the **DOES>** run-time action, and the second "noop" will be replaced by "exec", which will execute that procedure:

```
/noop { } def

/create {
  [  here  /noop cvx  /noop cvx  ]
```

Remember the non-executable array as "lastacf" so that we can replace the noop's later if we need to:

```
dup /lastacf -def
```

Make the array executable and bind its procedures:

```
cvx bind
```

Read the name from the input stream and associate it with the new procedure:

```
bl word cvn  -def
  } def
```

We need a flag which tells ";" whether or not there is a **DOES>** clause:

```
/indoes false def
```

The run-time action executed by defining words modifies the child word so that it executes the **DOES>** clause:

```
/dodoes { lastacf 1 rot put lastacf 2 /exec
  cvx put } def
```

From now on, we define immediate words, placing them in the "immeddict" dictionary:

```
immeddict begin
```

A **CREATE .. DOES>** defining word is compiled as follows:

★ : foo CREATE a b c d DOES> e f g ;
★ ==>
★ /foo { a b c d { e f g } dodoes } def

```
/does  { +level  /indoes true def  }  def
```

Semicolon ";" has to know whether or not there was a **DOES>** clause, so it can decide whether or not compile "dodoes":

```
/;  {
  indoes  { /indoes false def  -level
  (dodoes) cvx  }  if -level def
  }  def
```

IF .. ELSE .. THEN is really quite easy. It compiles as follows:

```
★   :foo a b IF c d THEN e f ;
★   ==>
★   /foo {a b {c d} pif e f} def
★   :foo a b IF c d ELSE e f THEN g h ;
★   ==>
★   /foo {a b {c d} {e f} pifelse g h} def
```

The reason we have to compile "pif" or "pifelse" instead of the normal PostScript "if" and "ifelse" is because the PostScript operators require a boolean type operand, whereas Forth uses numbers for flags. The "nip /pifelse cvx" in "else" replaces the "pif" left by "if" with "pifelse":

```
/if   { /pif cvx +level  }  def
/else { -level nip /pifelse cvx +level }
  def
/then { -level swap }  def
```

Other Forth conditionals are left to the reader as an exercise. Hint: Use "loop" and "exit" to build **BEGIN .. WHILE .. REPEAT** , **BEGIN .. AGAIN** , and **BEGIN .. UNTIL** . Use "for" to build **DO .. LOOP** and **DO .. +LOOP** , being careful of boundary conditions!

Mitch Bradley is a well known engineer (with Sun Micro-Systems) and a fluent programmer in Forth, PostScript and C. Through his alter ego, Bradley Forthware, Mitch offers a Forth written in C ("CForth") which is Forth-83 compatible and runs on Unix / Xenix / Minix. He also is a member of the ANSI Forth Standards team.

❑❑❑

Software Availability

The executable portion of the above article can be downloaded from the Forth Interest Group area of GEnie, as filename "PSFORTH.02.90".

Don't miss the 4th Annual SIGForth Workshop to be held March 5-7, 1992 in Kansas City!

ANNOUNCEMENT!

The 1990 and 1991
ACM SIGForth Workshop
Combined Proceedings
will shortly be available.

Papers from both meetings are included in this one volume for your convenience!

Prices:
$14 (ACM and SIGForth Members)
$20 (Non-Members)

Contact ACM for your copy now!

NEWS FLASH!

What's happened to Harris 's RTX-2010 Forth engine project?

(Dateline: Melbourne, FL) According to information given to your Editor by well-placed corporate sources, the RTX-2010 has been made into a rad-hard part and given to Harris's Military and Aerospace Division. The new designation is "RTX-2010RH" for this 1.2 micron geometry processor in CMOS (silicon-on-sapphire).

Using only 4100 gates, the RTX cpu design is expected to be carried in the Harris Macrocell Library for another 10-15 years. No off-the-shelf parts will be available, as each chip is to be produced as a one-off ASIC for customers. This part has a 30:1 price / performance advantage over the nearest radiation hardened alternative, the Mil-Std 1750A, according to Harris.

The first rad-hard silicon was produced and tested during mid-1991. It was able to tolerate 1,000,000 rads total dose and brief transients of up to 100,000,000,000 rads per second.

Software will include both Forth and C. Programming support will be through third party suppliers and the user community rather than via Harris.

Price (including NRE's) and availability can be obtained by calling Harris directly.